NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

OBJECT ORIENTED DESIGN
OF TACTICAL TIC-TAC-TOE
C4I SIMULATION

by

Todd L. Lennon

June, 1996

Principal Advisor: Gary Porter

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE OBJECT ORIENTED DESIGN OF TACTICAL TIC-TAC-TOE C4I SIMULATION

5. FUNDING NUMBERS

6. AUTHOR(S) Todd L. Lennon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The purpose of this thesis is to redesign the Tactical Tic-Tac-Toe (T4) game using object-oriented design. T4 is a C4I simulation developed by Prof. Gary Porter that is based on the traditional Tic-Tac-Toe game. It allows players to play against other players or against the computer. Various board sizes, multi-board games, delayed intelligence, team play, and limited communications are used to model real world C4I problems. The game allows for data collection for later analysis of game configurations and results. The goal of this thesis is to redesign the original program written in Macintosh HyperTalk language by using the Booch object-oriented design method and the C++ programming language for porting the program to a Unix or Windows environment with the ultimate goal of having a networked game that can be played remotely using a WWW browser type interface.

This design used requirements analysis and domain analysis to create class, operation, and attribute definition. Class association, aggregation, and inheritance are also specified. This design is ready to begin control class definition, access control definition, and operation algorithm development in preparation for coding an executable release.

14. SUBJECT TERMS T4, TIC-TAC-TOE, C4I, GAME, OBJECT, ORIENTED, DESIGN, SIMULATION

15. NUMBER OF PAGES 104

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

OBJECT ORIENTED DESIGN OF
TACTICAL TIC-TAC-TOE
C4I SIMULATION

Todd L. Lennon
Lieutenant, United States Navy
B.B.A., Texas A&M University, 1988

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY
(Command, Control, and Communications)

from the

NAVAL POSTGRADUATE SCHOOL
June 1996

Author: Todd L. Lennon

Approved by: Gary R. Porter, Principal Advisor

Michael G. Sovereign, Assistant Advisor

Dan C. Boger, Chairman C3 Academic Group

ABSTRACT

The purpose of this thesis is to redesign the Tactical Tic-Tac-Toe (T4) game using object-oriented design. T4 is a C4I simulation developed by Prof. Gary Porter that is based on the traditional Tic-Tac-Toe game. It allows players to play against other players or against the computer. Various board sizes, multi-board games, delayed intelligence, team play, and limited communications are used to model real world C4I problems. The game allows for data collection for later analysis of game configurations and results. The goal of this thesis is to redesign the original program written in Macintosh HyperTalk language by using the Booch object-oriented design method and the C++ programming language for porting the program to a Unix or Windows environment with the ultimate goal of having a networked game that can be played remotely using a WWW browser type interface.

This design used requirements analysis and domain analysis to create class, operation, and attribute definition. Class association, aggregation, and inheritance are also specified. This design is ready to begin control class definition, access control definition, and operation algorithm development in preparation for coding an executable release.

TABLE OF CONTENTS

EXECUTIVE SUMMARY

The purpose of this thesis is to redesign the Tactical Tic-Tac-Toe (T4) game using object-oriented design. T4 is a C4I simulation developed by Prof. Gary Porter that is based on the traditional Tic-Tac-Toe game. It allows players to play against other players or against the computer. Various board sizes, multi-board games, delayed intelligence, team play, and limited communications are used to model real world C4I problems. The game allows for data collection for later analysis of game configurations and results. The goal of this thesis is to redesign the original program written in Macintosh HyperTalk language by using the Booch object-oriented design method and the C++ programming language for porting the program to a Unix or Windows environment with the ultimate goal of having a networked game that can be played remotely using a WWW browser type interface.

This design used requirements analysis and domain analysis to create class, operation, and attribute definition. Class association, aggregation, and inheritance are also specified. This design is ready to begin control class definition, access control definition, and operation algorithm development in preparation for coding an executable release.

Object-oriented design can be used to redesign a non-object-oriented game program as evidenced by this thesis.

A greater level of detail is required than was reached in the thesis to begin coding the program. More detail in operation and attribute specifics will be required.

An attempt at building in flexibility for future evolution of the software was made. For example, a double game board under the current requirements only consists of two three-

by-three game boards positioned side-by-side. The game board class however has the flexibility to accept different size boards with a border positioned anywhere the user specifies. As long as all of the game algorithms are designed with general rules which do not depend on a specific size, this flexibility will remain in the design.

The current version of HyperText Mark-up Language (HTML) does not provide the functionality to permit use of a World Wide Web (WWW) browser as a networked game interface because it does not allow the host game engine to prompt the player for action. It is suspected that the Java language will provide this functionality and that the WWW browser will be an effective interface for the networked game.

The Booch method proved to be a very usable method for object-oriented design. This method and object-oriented design in general are not as intuitive as many texts imply, but with a good tutorial and a little experience with an object-oriented language such as C++, a designer can successfully create a viable object-oriented design. For the beginner, a good CASE tool is highly recommended, if for nothing else than enforcing the rules for a particular design method.

The T4 design in this thesis is not at the stage to begin coding the final executable program. The design is ready for the final steps of the Booch method which include determining access control, adding control classes, and defining algorithms to implement all of the operations. A firm foundation has been established for continuation of the T4 migration project.

It is recommended that the design be continued using the Rational Rose C++ CASE tool. This CASE tool provides functionality for configuration management, team

development, code generation, and reverse engineering.

Future versions of the T4 game should include some basic analysis capability to allow remote users to benefit from the educational opportunities provided by the game. Future versions should also consider portability to PC platforms as they take over more of the marketplace previously dominated by high power workstations.

Finally, the T4 game should be made available on the Navy Online web site to maximize participation by, and education of, all DOD members.

I. INTRODUCTION

A. BACKGROUND

The Naval Postgraduate School (NPS) offers the Command Control and Communications (C3) Systems Evaluation course (CC4103) during the fall quarter. It is primarily intended for students in the Joint Command, Control, Communications, Computers, and Intelligence (JC4I) Systems curriculum and, as such, is provided in their fifth (of seven) quarters. The course is designed to be one of the curriculum capstone courses. C3 experiments related to the C3 evaluation process are planned, conducted and analyzed during this course.

Several C3 related experiments are conducted in support of the course to reinforce classroom theory and to provide hands-on experience to the students. In each experiment a different group of students is assigned as the lead group. These students are responsible, at least in part, for all phases of the experiment including design, conduct, and analysis of the experiment. Normally the first experiment in the course is the simplest in order to set the stage for the more complicated experiments that follow.

A tactical version of the Tic-Tac-Toe (T3) game called T4 was developed to fulfill the need for the first experiment used in the C3 Systems Evaluation course because it:

1. Is easy to learn
2. Is new to all subjects
3. Is a non-military oriented game
4. Generates data that are C3 related

T4 is a multiplayer, networked game designed by Professor Gary Porter to model the effects of different levels of C4I on game outcome. The game is based on the traditional game of Tic-Tac-Toe but uses event timing, information hiding, assignment of mission objectives, and team play to simulate the effects of poor communications, uneven combat power, intelligence deficit, and tactical style. The game is currently written for a Macintosh computer in HyperTalk which is the HyperCard script language.

B. PURPOSE

The purpose of this thesis is to redesign the T4 game using the object-oriented paradigm by providing diagrams and specifications for recoding of the model. The ultimate goal is to reprogram this software in the C++ language to facilitate migration of the model to UNIX and other platforms. The model will evolve to use World Wide Web (WWW) browsers as the primary user interface in a networked multiplayer environment. This will make multi-team, networked play of T4 available to users of the internet.

C. RESEARCH QUESTIONS

Can a non-object-oriented model be redesigned using object-oriented methods and reprogrammed in an object-oriented language for porting to other platforms? Are current computer aided software engineering (CASE) tools useful in migrating existing non-object-oriented programs to other languages? What level of detail is required to provide a programmer with the information required to begin coding? How can flexibility for future evolution of the software be built in to the present design? Do current WWW languages like HTML and Java provide the functionality to permit use of a WWW browser as a standard user interface for a networked game? These questions will be answered in the thesis.

D. METHODOLOGY

In order to facilitate future upgrades, changes, multiple project team members, and portability across platforms, object-oriented design has been selected as the preferred design methodology before commencing the coding phase. In particular the Booch method of object-oriented design will be used. The Booch method was developed by Grady Booch of Rational Software and will be covered in detail in Chapter III. The first step in the design will be to examine the existing modular design and isolate functional and data requirements. Functional and data items will be organized in an object-oriented hierarchy maximizing inheritance and code reuse. The design will attempt to anticipate future data and function requirements and provide for expandability in the design. Object-oriented breakdown charts will be used to describe inheritance relationships while actual object specifications will be described in detail in the text of the thesis. CASE tools will be used where appropriate to aid in the development of the design.

E. ORGANIZATION OF THESIS

This chapter discussed the general approach of this thesis project. Chapter II will contain a discussion of the current version of T4, including the history and the details of how the game is played. Chapter III gives an overview of object-oriented design principles along with a detailed treatment of the Booch method of object-oriented design. Chapter IV covers the process of the actual T4 design project and is broken down along lines similar to those of Chapter III. Finally Chapter V contains a summary of the T4 project along with conclusions and recommendations.

II. TACTICAL TIC-TAC-TOE

This chapter discusses the history of T4, the genesis of the migration project, the T4 baseline game, its major features, double board games, the intelligence delay factor and multiple players, the networked game, and finally the closed-form simulation mode. The T4 game described here is the current version and serves as the basis for requirements for the design undertaken in this project.

A. HISTORY OF T4*

T4 was purposefully developed to serve as an experimentation device for the CC4103 Systems Evaluation course. The goal was to minimize training time in using the game while maximizing the systems evaluation experience gained while using the game for practical experiments. Development was begun in 1990 by Gary Porter. T4 was chosen because it is a simple-to-learn game, similar to T3 to the extent that previous play or experience does not tend to be a noticeable factor in the game outcomes after a player has played very few games. The learning curve for T4 is so steep that the advantage of experience is quickly overcome. This is not the case for all simulation games, chess for example, has rules that are fairly easy to learn but experience is a major factor in outcome performance. It is very difficult to design experiments that can factor out the effects of experience. A more logical solution is to choose a game in which experience has little effect. An added advantage of T4

---

*Most of the description of the T4 game in this chapter is paraphrased from an unpublished document written by Professor Gary Porter.

is that the common knowledge of the rules of T3 tend to reduce training time for the T4 game.

HyperTalk, the scripting language for Macintosh HyperCard software was chosen as the programming language because of its rapid development capability for an interactive graphical user interface. The program was written using a traditional structured programming style. Gary Porter has been the sole developer and programmer on the project.

Development of the single player version and the team player version of the game was completed and used in 1991. Students observed that the single player version of the game did not provide enough relevant data for study and immediately moved to the team player version for the class projects. These versions of the game were played on one machine with game play conducted by allowing players to view printouts of the current game board and then submitting their moves on paper for entry into the game program by game controllers. This form of play proved cumbersome, time consuming, and error prone. In 1992 a networked version of the game was introduced with the hope of increasing the speed of the game play. Game play was only speeded slightly but administration of the game was eased significantly as was the amount of transcription errors.

Because of these problems, the most popular version of the game is the closed simulation mode where various game configuration are input and simulated players play the game and produce outputs to data files for subsequent analysis. The closed simulation mode is discussed in more detail at the end of this chapter.

Drawbacks to the current design include speed, user interface, and fault tolerance. Because the program is written in a translated scripting language, the speed of modern

processors is not utilized as it would be if the game program were a compiled executable program. The user interface needs to be improved to allow game players to play independently with a minimum of supervision. The interface has to be more apparent and user friendly. The game also has to become more fault tolerant if independent play is to become a reality. The game has to be able to detect all player errors and recover from them without aborting the program execution or producing erroneous data. The game was last used to support CC4103 in 1994.

B. GENESIS OF THE MIGRATION PROJECT

This section discusses the motivation for the T4 migration project. It covers Navy online war gaming initiatives, the need for an experimentation engine for CC4103, the requirement for an open systems design, and the desire to use the WWW as the network interface.

1. Navy Online War Gaming

The Director, Chief of Naval Operations Strategic Studies Group, located at the Naval War College has initiated an outreach to establish, through the offices of the Center for Naval Analysis, a home page for innovative naval warfare concepts on Navy Online. Navy Online is a World Wide Web server administered by the Naval Computer and Telecommunications Station in Pensacola Florida. Navy Online, serves as a gateway to United States Department of the Navy online resources. The second step of this initiative is to involve Navy members in war gaming using Navy Online.

The CNO Strategic Studies Group has three types of games in mind. The first type is abstract games that are designed to address theoretical issues. Tactical Tic-Tac-Toe has

been identified as an example of this type of game. The second type of game is the standard theater level game that looks at a variety of contingencies. These games would be played by teams who form command structures and submit operational orders to their forces using actual command procedures, modified only for security concerns. The third type would be games targeted specifically at evaluating the ideas proposed by concept generation teams. Analysts working with the teams would create new models need to evaluate operational concepts, and implement them on computers for analysis and gaming.

2. Experimentation Engine for CC4103

The T4 game was originated to fulfill a need for a basic experimentation engine for the C3 Systems Evaluation course. The need was for an easy to understand, easy to use, modifiable model that allowed for adjustment in several game environment factors. The game also had to provide the capability to run in a closed-form simulation mode (computer playing itself) to provide large data sets for the course's experimental analysis purposes. The original T4 model accomplished all of these goals. The need still exists for a CC4103 experimentation engine. The new design will improve on the above characteristics, concentrating on usability in order to maximize the student interaction with the model and not on the model's documentation. The new design will also attempt to embody the object-oriented qualities of portability and extensibility.

3. Open Systems (UNIX)

Migrating the T4 game to a UNIX platform using C++ with a WWW interface is an attempt to maximize the portability of the model. The UNIX platform was selected because the power, availability, and open systems architecture of UNIX-based workstations has been

required in the past for distributed gaming systems. Although the advances in IBM PC type platforms is making this less and less the case, it is believed that porting the game to a UNIX platform will make the game more universally usable today in the global, multiplayer, internetworked environment. Selection of the C++ programming language along with an object-oriented design will make the process of porting the game to a platform other than UNIX even easier in the future. C++ is well supported by design tools and is widely accepted in the programming industry. C++ development packages are readily available for many different operating systems and platforms.

4. WWW for Network Interface

One problem with producing a game designed for client/server, networked play among users with many different types of computers is the design of a common user interface. This used to be a daunting if not impossible task.

Now, there is an almost universally accepted client/server interface in place on almost all machines in the world that have internet access. This interface is commonly called a World Wide Web browser. There are many common brand names for WWW browsers such as Netscape and Mosaic. WWW browsers interpret and present information to a user based on a common scripting language called HyperText Mark-up Language (HTML) and a common signaling protocol called HyperText Transfer Protocol (HTTP). A new scripting language developed by Sun Microsystems, called Java, along with the capabilities of HTML, are now robust enough to support a fully interactive user interface. Java allows client interface programs to actually be transferred to the local machine where it is compiled and run on the local WWW browser. This promises true interactive, platform independent, user

interface for networked game play. Java is still evolving and currently not all WWW browsers support it. However, it is expected that popular WWW browsers will move toward full Java compliance.

The WWW interface is a practical solution to providing a common user interface and satisfying the requirement for universal usability. WWW browsers can allow game play on networks that don't have WWW access. A local area network (LAN) only has to have the game server and a WWW server software installed to allow local networked game play.

C. T4 BASELINE GAME

This section discusses the rules for the T4 game. T3 rules are first described, then the differences between T3 and T4 are covered with subsections on simultaneous moves, conflict resolution, and scoring.

1. T3 Rules

Tic-Tac-Toe is played with two players on a three by three grid. One player's moves are represented by Xs and the opponent's moves are represented by Os. Players alternate turns marking one of the grid cells with their symbol. The first player to get three of their symbols in a row (TTT), either horizontally, vertically, or diagonally wins the game. Tie games (Cat games) are possible and are actually the most likely outcome with experienced players since it is always possible to tie a game unless a player makes a mistake. The T4 game expands on this basic set of rules as described in the following subsections.

2. Simultaneous Moves

T4 players move simultaneously instead of taking turns as in the T3 game. Both players choose their moves in secret and submit them to the game program. The game program then processes the moves simultaneously and returns the results.

3. Conflict Resolution

The simultaneous move rule means that two players can attempt a move into the same cell on the same move. The winner of this conflict is randomly resolved. The winner of the conflict takes the cell and the looser gets no cell, effectively loosing a turn. The resolution of the first conflict has a great impact on the outcome of the game. Note, the final move of the game is always a conflict. Only a single cell remains and therefore requires a conflict resolution.

4. Scoring

In T4 the player with the most TTTs at the end of the game wins (not the first TTT). This means that play continues until all nine cells are filled. There are eight TTT possibilities (three horizontal, three vertical, and two diagonal TTTs). Therefore the maximum player score is eight and the minimum player score is zero. Note that, in order to score eight TTTs, all moves must result in a conflict and the same player must win every conflict. The probability of this occurring is about the same as winning the Lotto (approximately 1 in 20,000,000). Variants to the basic T4 games are described in the following sections.

D. DOUBLE BOARD GAMES

The game boards in a double board game are two single game boards connected side-by-side. A double game board game is equivalent to playing two baseline games simultaneously with the added feature of crossover TTTs. A crossover TTT occurs when three of the same symbols (either Xs or Os) are in a straight line and at least one of the symbols in this line occupies a space on each board, i.e. the TTT line crosses the border between the single game boards. Double board games can be played by single players, and they are a requirement in team games.

E. MULTIPLE PLAYERS

This section covers the basic two player T4 game and the T4 team game. It also covers mission assignment and scoring.

1. Two Players

The basic game is played with two players. Two player games can be played with all variants of the game as well as with the different mission assignments as will be discussed later. Two player games can be conducted with the three possible combinations of real and artificial players.

2. Teams

Team T4 games consist of four players in teams of two for each side (X and O). One player on a team is assigned the left side of the game board and the other the right side. Team games are always played on double game boards with mission assignments.

3. Missions

Missions are assigned to teams before game play begins. The players on a team are

always assigned the same missions; however, different missions may be assigned to opposing teams. Mission assignments are not divulged to the other team. Mission success is determined by counting own and opponent TTTs in the designated mission areas. Two general types of missions are victory and survival missions. A side must score more TTTs in a mission area than the opponent to be considered victorious in that mission area. A side must only match the opponent's TTTs to achieve survival in a mission area. Up to eight different types of missions may be assigned to a team as follows:

Victory by Game Board Area

1. **VL** = The most TTTs (victory) on the left side of the double board game.
2. **VR** = The most TTTs on the right side.
3. **VC** = The most cross over TTTs.
4. **VO** = The most total TTTs (left, right, and crossovers).

Survival by Game Board Area

5. **SL** = Don't lose (survival) on the left side.
6. **SR** = Don't lose on the right side.
7. **SC** = Don't lose in the crossover area.
8. **SO** = Don't lose overall.

Up to four individual missions may be assigned to a team's mission set per game (players on the same team are always assigned the same missions).

4. Planning

Team games can occur with three different levels of planning, specific planning, general planning, and no planning. Specific planning is when missions are provided and direct conversation is allowed before game start in order to plan the specific missions. General planning takes place when team members are assigned but before the game is configured so that planning for specific missions is prevented. No planning occurs when

players are assigned to teams at random immediately prior to game play with no conversation permitted until the end of the game.

5. Scoring

Scoring is based on successful mission achievements. TTTs are used to decide mission outcomes. Notice that both zero sum and non-zero sum games are possible. A zero-sum game results when both players are assigned the same victory missions. A non-zero sum game results when both players are assigned survival missions or different mission areas. It is therefore possible for both sides to meet the assigned mission objectives depending on the missions assigned.

F. INTELLIGENCE DELAY FACTOR

A T4 game played with intelligence delay is played like other games except that knowledge of other player's moves can be delayed from one to nine turns. A player is always aware of the outcome of his own moves. Also, a player is immediately aware of opponents moves that cause a conflict (opponent and own player move into the same cell). Three categories of information delay can occur and the amount of that delay can be assigned from zero to nine delay turns, independently adjusted for each of the three categories and for each of the four players. Intelligence delays are not divulged to any player or team including own player or own team. Delay conditions may be deduced during game play.

1. Tactical Delay

Tactical intelligence delay is a delay in information concerning own opponent's moves. The distinction of own opponent and partner's opponent will be made clearer in the later discussion on team play. Tactical delay may be set from zero (no delay, real-time

information) to nine (player never receives information about opponent's moves unless there is a conflict). For example, in a scenario with tactical delay set to two, a player would only become aware of an opponent's initial first turn move after the third move occurred. This raises the issue of delayed conflicts. A delayed conflict occurs when a player unknowingly moves into a cell that the opponent already occupies. This situation may occur due to the tactical delay feature. In this case the player who occupies the cell first always wins the conflict and both players are informed of the outcome. Tactical delay can occur in single player games or team games.

2. Area Delay

Area intelligence delay involves the delayed knowledge of moves made by a partner's opponent. This information becomes critical in double board games with crossover mission objectives as was discussed in a previous section. Delay levels are only overridden when own player is involved in conflict resolution. Since area delays and communication delays (to be discussed next) do not directly involve own players, delays in these two categories are never overridden. Area and communication delays can only occur in team games.

3. Communications Delay

Communications delay controls the timeliness of receiving a partner's move information. Like tactical and area delay, communications delay can be set to real time (no delay) or a delay of one to nine turns. Communications delay can only occur in a team game.

G. NETWORKED GAME PLAY

A goal of the T4 project is a networked gaming system. The networked game may be played by players separated geographically. Geographic separation should not be apparent to the players. A host machine will run the game engine and allow players to log on using internet protocols over a local area network (LAN) or the internet via a WWW browser. Moves will be submitted to the game engine on the host machine and the results will be displayed on the individual client machines. The networked game will allow for participation by a greater number and a wider variety of players with a corresponding increase in the amount of data that can be collected in an experimental setting.

H. CLOSED FORM SIMULATION

Experience gained during original T4 simulation showed that while game play was fairly uncomplicated, the administration of the game by controllers was cumbersome, confusing, time consuming, and error prone. It is also manpower intensive when human players are the only subjects. Artificial intelligence modules simulating human game play in the program obviate the need for human subjects. This allows for rapid data collection and accumulation of large data sets using simulated players. These modules will also allow experimenters to fine-tune game factors for human players to assure that data is collected in the area of interest without wasting the time or interest of the human subjects. Simulated players may also be used in any combination with human players as well.

1. Game Plans

In the closed form game, each simulated player is controlled by its own set of three game plan matrices. These simulated player game plan matrices, assigned by the user before

a trial starts, control how the simulated players will attempt to achieve their assigned missions. These three game plans are the regular game plan, the crossover game plan, and the cell game plan. The regular and the crossover game plans enumerate, in coded form, all possible next moves for a simulated player. The user, by assigning point values (priorities) to these move options, in conjunction with the values (priorities) in the cell game plan, determine how a simulated player selects the next move. The cell game plan defines specific areas of the board that are of higher interest to the player.

2. TTT Tracks

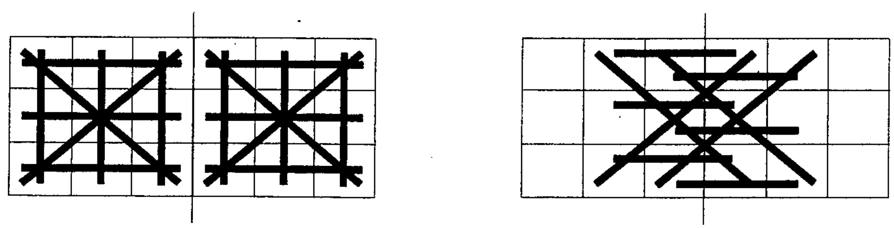
All of the possible ways of getting three cells in a row (Tic-Tac-Toes) are called the TTT tracks. TTT tracks include regular and crossover TTTs. Together they exhaust the possible TTT tracks on the team T4 game board. Regular TTT tracks are associated with the regular game plan, while crossover TTT tracks are associated with the crossover game plan. Regular TTTs are based on the eight conventional ways of completing TTTs on a single board for each player. Crossover TTT tracks are defined as those TTTs which cross over the center line of the double T4 game board. There are ten possible crossover TTTs, therefore, there are twenty-six unique TTT tracks on a T4 double game board.

3. Game Plan Code

The regular and crossover game plans contain coded descriptions of potential TTT tracks in the left column and user assigned point values (move priorities) in the right column. The code in the left column consists of a string of three characters. Each character is either an "X" or an "O" which represents an occupied square on the game board or a dash (-) which represents an empty cell. The permutations of the three character codes in each row of the

left column of the game plan represent all possible combinations of characters in a potential TTT track that contains an empty cell. These exhaustively represent all next move options for the simulated player. Diagrams of the TTT tracks and the game plan matrices are shown in Figure 1.

8 Regular TTT Tracks **10 Crossover TTT Tracks**



Regular Game Plan

OO–	1,000
XX–	500
O––	100
X––	50
–––	10
OX–	0

Crossover Game Plan

OO–	1,000
XX–	500
O––	100
X––	50
–––	10
OX–	0

Cell Game Plan

2	3	5
1	4	5
2	3	5

Figure 1. TTT Tracks and Game Plan Matrices

4. Track to Code Match

The program systematically cycles through all eighteen double game board cells for each simulated player looking in each of eight possible TTT tracks for a sequence of three cells, of which one is blank. This represents a possible next move. Each potential next move is matched with a permutation of the code in the appropriate regular or crossover game plan

matrix. The value associated with the code is added to each empty game board cell in the TTT track. A cell with a point value already assigned to it is still considered an empty cell because it does not contain an "X" or an "O" and thus remains a candidate for selection as the next move. Notice that empty cells may accumulate points from more then one TTT track. After cycling through the eighteen cells and assigning regular and crossover point values to the empty cells, the program cycles through each empty game board cell and adds to it the points in the corresponding game cell plan.

5. Next Move Selection

The program then selects, as the simulated player's next move, the empty cell that has the highest accumulated point value. Ties are broken randomly. The point totals are erased and the same procedures are used again to select the next move for all simulated players. This simulated move procedure is a tedious and exacting process and is well suited for computer implementation.

6. Player Personalities

The user selects personalities for the simulated players by filling in the game plan matrices. These personalities can be offensive, balanced offensive, neutral, balanced defensive, defensive, or random. The personalities are effected by the values assigned to the possible permutations for characters in each TTT track. For example, if the user wanted player "X" to be defensive in nature, he would a higher value in the matrix corresponding to the (OO-) combination than the value corresponding to the (XX-) combination. In this way, the simulated player "O" would always choose to block player "X's" TTTs instead of completing its own TTTs.

III. OBJECT-ORIENTED DESIGN METHODOLOGY

This chapter explains the need for, and the general principles of, object-oriented design (OOD). Different OOD methods are mentioned with a detailed discussion of the Booch method which was chosen for this design project.

A. INTRODUCTION TO OBJECT-ORIENTED DESIGN

This section suggests complexity as the impetus for modern object-oriented methods. The object-oriented paradigm is covered with a brief discussion of object-oriented methods.

1. Complexity

The challenges facing software designers today can almost without exception be attributed to complexity. The complexity of the software is derived from the complexity of the problem domain which is derived from the complexity of our environment. The goal of the software engineer is to hide the complexity and create the illusion of simplicity for the user. In order to accomplish this goal, the complexity of today's software systems must be managed in a standardized methodical way.

Historically, the complexity of the software systems has driven the methods of software design (Booch, 1994). From the 1940s to the early 1960s, the programming community did not use any one systematic approach to software design. Machine capability limited the complexity of the software to a level manageable by one or a few programmers using a nonsystematic programming approach. Although this early software served its purpose, the program code was hard to follow, rarely reusable, and extremely difficult to debug. This style of programming was often referred to as "spaghetti code" because

outsiders were often unable to detect any recognizable logic pattern in the programs source code.

As the performance of computer systems increased, so did the complexity of the user requirements and the ability of software to handle those complex requirements. As the complexity increased, it became increasingly more difficult for programmers to manage the logical design of the software. These problems manifested themselves as schedule overruns, greatly exceeded budgets, and unreliable finished products. "Research activity in the 1960s resulted in the evolution of structured programming, a disciplined approach to writing programs that were cleaner than unstructured programs, easier to test and debug, and easier to modify" (Deitel, 1994). Structured programming is the style with which most programmers, who learned to program in the 1970s and 1980s, are familiar with. Structured programming is characterized by single-entry/single-exit control structures which are connected in a logical sequence to reflect the logic flow of the algorithm being designed. The rules for forming structured programs allow the control structures to be nested. Structured programming is language independent. Design can be accomplished prior to language selection. The individual language constructs of most higher level languages can then be used to implement the control structures in the design.

2. Object-Oriented Programming

The roots of object-oriented programming (OOP) first began to appear in the early 1970s (Deitel, 1994). Almost simultaneously, the term, object, as applied in computer science seems to have gained usage independently in various fields of computer science. The word, object, was first used to "refer to notions that were different in appearance yet mutually

related" (Booch, 1994). These notions were a method of managing the complexity of software systems where the objects represent components of the decomposed system. The object idea was originally applied to hardware components but naturally flowed into software applications, eventually. Object-oriented programming is a method that attempts to capture the behavior of the real world in a way that hides the detailed implementation (Pohl, 1993). This approach allows the problem solver to think in terms of the problem domain.

Any discussion of object-oriented programming begins with a definition of terms. OOP is a data-centered approach that attempts to link behavior with data.

Data and behavior are grouped into classes that are instantiated as objects in the program. Besides the standard data types that are native to the language, e.g., integer, character, or floating point, an object-oriented language will allow the programmer to construct customized data types. The language also allows the programmer to determine the behavior and the way that the program handles these data types. These data types are abstractions of real world cases and are referred to in object-oriented terminology as abstract data types (ADTs). A class is an aggregation of data and the behavior associated with that data. In the real world, a class could be a species of animal such as a dog. Dogs have certain characteristics and behaviors that can be defined. A specific instance of a class is an object; for example, Fluffy could be an instantiation of the class dog. Fluffy would have all of the characteristics and behavior that are defined in the dog class but is specifically identifiable as one instance of a dog named Fluffy. Sparkey could be another instance of the dog class. Fluffy and Sparkey would be separate items with similar characteristics. In the context of an OOP program, Fluffy and Sparkey would be objects. They would represent an ADT that

is specified in the class dog. Fluffy and Sparkey are akin to variable names of the native data types and are used in a similar manner.

Another key aspect is that of inheritance. Inheritance is the capability to derive a new type from an existing type. Using inheritance, the new type can take advantage of some, or all, of the previously defined attributes and behaviors. The new type can create new attributes and behaviors and can even redefine the inherited attributes and behaviors. Referring back to the earlier example, a more general class called mammal exists. Mammals can be defined by their attributes and behaviors. There are many types of mammals and it would be wasteful and repetitive to redefine all of the attributes and behaviors that are common to all mammals for each individual class of animal. What makes more sense is to define these traits only once in the class mammal. In this way, the dog class only has to specify its inheritance from the mammal class to gain access to all of the mammal traits. The only new things that have to be added are dog specific. Inheritance can be passed through many layers of classes. For example, the mammal class could inherit from the more general animal class and a schnauzer class could inherit all of the traits of the more general classes above simply by claiming inheritance from the dog class. Inheritance is often expressed in plain language terminology as an "is a" relationship. A schnauzer "is a" dog, is a mammal, is an animal, and so on.

Aggregation is another characteristic of object-oriented programming. Aggregation is the capability to use one type as a component of another type. This is different from inheritance. The aggregate type uses the subtype as a building block. This quality is described in plain language terms as a "has a" relationship. A string data type can be

considered to be an aggregate of the character data type. A string "has a" character. In fact, a string normally has several characters. This is quite distinct from inheritance. A string does not inherit the traits of a character but is a collection of several characters. Aggregation, by allowing complex types to be built from simple types is another way that object-oriented programming facilitates code reuse.

A final aspect of object-oriented programming to be addressed is encapsulation. As mentioned earlier, OOP attempts to package data and behavior together. This encapsulation is a requirement for being able to implement abstract data types. When a new data type is created, mechanisms for manipulating these data types must also be created. These functions and operators are defined within the code of the data type. This means that whenever an ADT is declared and used in a program, all of the necessary function and operator definitions come with it and do not need to be redefined. The ADT is a whole package in object-oriented terms. This is the concept of encapsulation.

An advantage of encapsulation is data hiding and the hiding of the implementation detail. The programmer uses the ADT and manipulates it via its publicly available functions and operators but does not need to be aware of how the functions are implemented nor of any hidden data types used in the implementation of the functions. In fact, this principle prevents the programmer from having direct access to the data at all. The data should only be accessed through the public interface of the functions and operators. This aspect provides safety and enhances reusability. Once an ADT has been created, tested, and debugged, it becomes a trusted building block for the programmer who only needs to be concerned with how to use the public interface and not with the inner workings of the ADT.

3. Specific Object-oriented Methods

OOP is a generic concept that is characterized by very broad principles. There are many very specific object-oriented methodologies that provide a step by step procedure for going from the concept development phase to producing the final executable program product. Many of the methods vary greatly. Examples of some of the current object-oriented methodologies are the Booch Method by Grady Booch, Object-oriented Modeling Technique (OMT) by James Rumbaugh, Object-oriented Software Engineering (OOSE) by Ivar Jacobson, and Syntropy by Daniels and Cook. The choice of an object-oriented method is largely dependent upon personal preference but the factors that could affect the choice are practical usability, documentation, Computer Aided Software Engineering (CASE) tool support, extensibility to a specific programming language, and a general acceptance by other project participants. Probably, the most important factor is whether the method will support your project completely from beginning to end. As a side note, Booch, Rumbaugh, and Jacobson are currently collaborating at Rational Software to create the Unified Modeling Language method, an open, industry-standard method evolving from the best aspects of all three methods.

For this project, the Booch method was selected. The Booch method has been criticized for its overuse of symbols and verbiage but one of the strengths of the Booch method is that, because of the rigor of the method, the end product is a well-designed system specification. The Booch method is well documented and is widely accepted in the software industry. The Booch method is also well supported by the Rational Rose CASE tool by Rational Software. The particular version of the CASE tool used in this project, Rational

Rose/C++ version 3.0, actually expedites the software coding by generating C++ source code based on the model constructed with the CASE tool.

B. INTRODUCTION TO THE BOOCH METHOD

The Booch object-oriented method is used to develop a system design that will primarily be implemented in software. "The Booch method is an object-oriented method based on a proven heuristic for developing quality software that not only provides an effective design but also supports that design and the development of future systems" (White, 1994). Use of the Booch method requires developers to produce a model of the system. The model will be used directly in the formulation of working code and executable releases. The Booch models are built in stages that allow focus on specific aspects of the system at a given time.

This method is intended to be an iterative approach to software design. All of the discovery that takes place in understanding requirements, services, devices, and system stages cannot take place during discreet intervals of the design process. The Booch method supports this iterative process; it allows for stepping through the process, discovering as you go, and then going back to make changes and trying it all again. The steps, in practice, are a guideline that is applied iterativly to the design process as the system gets increasingly more refined.

The three basic steps of the Booch method are requirements analysis, domain analysis, and system design. Requirements analysis provides the basic charter for the systems functions. Domain analysis provides the key logical structure of the system. System

design provides the key physical structure of the system, maps the logical structure to it, and leads ultimately to the working executable release.

C. REQUIREMENTS ANALYSIS

Requirements analysis is the phase where the designer attempts to understand as completely as possible, the needs of the customer. During this high level phase, key functions that the system is to perform are identified. The scope of the problem domain that the system will support is also identified and the key practices and policies of the organization using the system are documented. During this phase the designer works with the users and domain experts to determine particular use cases where the system will be applied. This is the first step in defining classes and operations, and will be used later in validation and testing.

One of the goals of requirements analysis is to form an agreement between the developer and the customers as to what the system will provide to the customer. These agreements are not fixed and as often is the case, new requirements arise or are discovered as the development process progresses. It is important though, to give requirements analysis proper consideration. Defining requirements as accurately as possible in the beginning can avoid many unnecessary costs in time, manpower, and money later on in the process.

The two products of the requirements analysis phase are a system charter and a system function statement. The system charter outline the responsibilities of the system [Appendix A]. The system function statement outlines the key use cases (an object-oriented term meaning situations where an object is used) of the system [Appendix B].

D. DOMAIN ANALYSIS

Domain analysis is the process of defining the aspects of the real world problem domain that will affect the system. The problem domain is the part of the real world that the system will interact with. This analysis identifies the data and major operations (classes) that will be needed to carry out the system's functions. It also defines relationships between classes, operations that those classes perform, their attributes including inherited properties, and validation and iteration.

1. Defining Classes

Since the major purpose of domain analysis is to identify the classes from which the system will be constructed, a method for identifying candidate classes needs to be chosen. The system designer normally has a good deal of knowledge about the object-oriented design process but usually has a limited amount of knowledge of the problem domain. Customers normally possess this expertise. This is why collaboration between the system designer and the customer is critical in this phase of the project. A good method for beginning to identify candidate classes is to develop a narrative system description. The narrative describes what the system does, the products it generates, and the scenarios in which it is used. Identifying all of the possible scenarios in which the application will be used is essential in understanding what tasks the application will perform. From the narrative, key classes can often be identified by underlining the nouns. This technique is useful because nouns often correspond to classes.

There are some problems when using this approach. The narrative often contains implementation characteristics while we are initially interested only in the logical classes.

It may also contain contextual information that is irrelevant to the system's responsibilities. The natural language of the narrative can contain terms that are ambiguous. The nouns in the narrative could not only be classes but could also be objects, relationships, or attributes of classes.

To separate key classes from these candidate classes, additional steps should be taken. Examine the tangible things and the roles that these things play in the system. Outline the steps necessary to complete the use listed in the system function statement. Identify the objects that participate in a functional scenario. Identify the responsibilities of each class, the knowledge that the class maintains, and the actions it provides. List the classes that collaborate with it to support these responsibilities.

Meaningful names should be chosen for each class and should reflect the part of the domain which it represents. Good names are simple and provide significant semantic information. The name should bring to mind the abstraction represented by the class.

2. Defining Relationships

Classes in the model do not exist independently. Defining relationships identifies the interactions that the classes exhibit. There are two type of relationships between classes, association and aggregation. Association indicates some kind of semantic dependency between classes. Aggregation, as mention earlier, denotes a "part of" relationship.

Associations are bidirectional relationships but they do not have to be implemented in both directions. Navigation can be restricted so that the association is only traversed in one direction. In early analysis it is sufficient to say that an association exists. The navigation paths can be refined as the analysis progresses.

Aggregations express "whole/part" relationships where the whole owns its part objects. Aggregation occurs when an object is physically constructed from other objects such as a house that is constructed of bricks, or when an object logically contains another object such as a homeowner having a telephone number. Ownership of the part by the whole is enforced by aggregation.

Like classes, the relationship should be given a meaningful name that expresses the nature of the semantic relationship. The relationship name often describes the way that the relationship is traversed and also the relationship of the target class to the source class.

Relationships are also described by their cardinality. Cardinality expresses information on quantity. A relationship can be mandatory or not. This is the minimum cardinality. A relationship can also have a maximum cardinality. If the relationship is bidirectional, then cardinality is expressed in both directions. An example of a bidirectional one to many relationship a father/children relationship. A father can have many children, but an individual child only has one father. Here, cardinality is bidirectional because it is expressed in both ways, father to child and child to father.

3. Defining Operations

After the classes and relationships have been identified, the operations that the classes perform must be defined. Operations can be chosen by forming scenarios from the system function statement and determining the operations needed to carry out those scenarios or by examining each class individually and determining which operations are required.

There are several guidelines that should be followed while designing operations. Each operation should perform one simple function. A simple function is sometimes called

a primitive. Having too many inputs and outputs can be an indicator that the function could be separated into separate operations. Input switches can also be an indicator that a function is not primitive and should be broken down into more simple functions. Each operation should be identified with a specific name that reflects the output of the function and not the steps that the function performs.

Operations are added to the class specification along with any information that is known about the arguments for the functions they are to perform. If the arguments for the operations are unknown, they can be added later. In the Booch method, object-scenario diagrams are used to trace how objects collaborate to perform a certain use case. These diagrams model the execution of a scenario. The diagram should use numbers to indicate the sequence in which the operations will be invoked to execute a scenario.

4. Defining Attributes

Attributes are also properties that describe a class. They can be classes, structures, or native data types that are required to implement a class. An attribute is equivalent to an aggregate association. The type of the attribute defines the general nature of the attribute. An example of an attribute for a hypothetical employee class in a payroll program could be "employee number". "Employee number" could be defined to be an integer where "employee number" was an instantiation of integer.

Two approaches to isolating attributes are to choose a class and list the properties which are attributes or to choose properties from sample output, input, or the problem statement and then determining what it describes. In order to identify a complete set of attributes, both methods must be applied. Finding attributes that cannot be associated with

an existing class may indicate the presence of a previously undiscovered class. Attributes are often discovered in the requirements documents. Adjectives that can be used to describe objects can correspond to attribute values. Attributes may also be derivable from other attributes by a function or operation. Here, a decision must be made to calculate the attribute when it is required or to use a persistent data item that represents the attribute.

5. Defining Inheritance

Sometimes similar classes have enough characteristics in common to be organized as a superclass and a set of subclasses. It makes more sense from a code reuse perspective to group similar traits into a superclass and to only specify various individual traits in the subclass. This procedure is the process of defining inheritance relationships. It is wasteful and introduces needless configuration management problems to duplicate common characteristics among classes. When defining inheritance relationships, it is important to remember not to confuse a reused type with an attribute that applies to more than one class. Often times a type can apply to more than one class where inheritance does not really make sense. For example, a date could be associated with a person's birth or when that person's drivers license expires, but it doesn't really make sense to inherit this characteristic from one superclass.

All of the classes and relationships described above are represented in the Booch method by class diagrams. An example class diagram is contained in Figure 2.

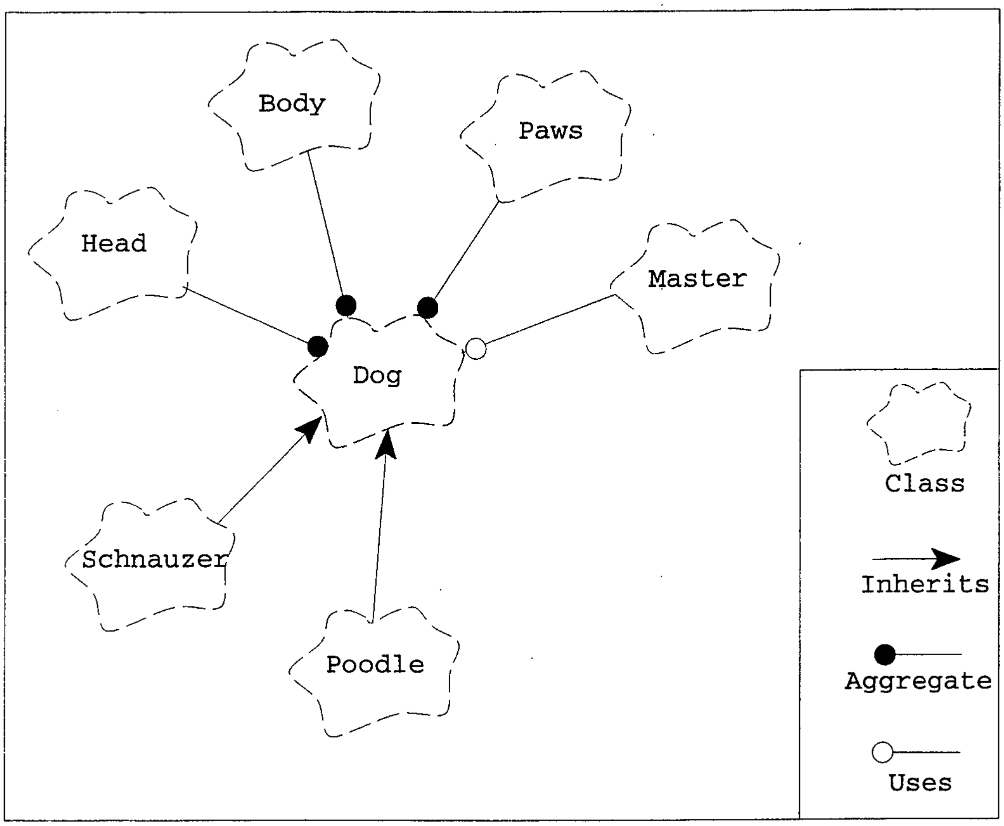


Figure 2. Example of Booch Class Diagram

6. Validation and Iteration

Recall that the Booch method is a highly iterative process. Throughout the domain analysis process and at each step specified above, there have been many opportunities to review, test, and repair this developing model as new aspects come to light.

A useful method for validating the model and providing more opportunity for iteration is to diagram the use cases for the system. These diagrams are called scenario diagrams. Examples of scenario diagrams are found in Appendix F.

Building scenario diagrams begins by choosing a simple process that the program must perform and deciding which class objects must participate in the performance of that process. When these objects have been identified, the messages that the objects must pass

to each other in the form of function calls are determined. The scenario diagram links the interacting objects and displays the message along these links. The messages are numbered so that a sequence of steps that are required to complete the process is defined.

This step provides an excellent opportunity to reveal previously unidentified attributes and operations in the model. This completes the discussion of domain analysis.

E. DESIGN

The third and final part of the Booch method is design. In the design phase, the effort is focused on how to implement the domain requirements discovered in the previous two phases. This section is not broken down into sequential steps. Rather, the design section covers general activities that are involved in the design phase.

Today's software is intended to have a much longer life than applications of the past. Over the course of its life, an application may be ported to other platforms, redesigned, evolved into new applications and new requirements, and may in fact exist in many different incarnations. A clean, well organized structure is much easier to understand, test, maintain, and extend.

A layered architecture is the basis of a good design. The lower layers need to provide the basic implementation for the layers above. The communication between the layers should be a one way process. Layers of the design use the services of the layers below but have no knowledge of the layers above. If the lowest layer of the system provides the logical interfaces to the hardware and supporting software then the system can be ported to other platforms simply by rewriting a single layer. Layers can be divided into loosely coupled

partitions that provide one kind of service. These partitions are called class categories in the Booch method.

The activities of the design phase are defining the initial architecture, planning the executable release, and developing the executable release. The products of a completed design phase should include a class category diagram, design class diagrams, complete design specifications, object-scenario diagrams, architectural descriptions, and executable release plans.

1. Initial Architecture

Defining the initial architecture involves partitioning the system classes into encapsulated groups. It also involves making design decisions about graphical user interfaces, database support, persistent object storage and peripheral services such as communications support. The choice is made to either build or buy these items but the interface to these functions remain a major portion of the design. Awareness of how the system will interface with these functions should be an initial design decision and should be chosen before getting into more specific implementation detail.

When adding all of the implementation classes to the design, the system complexity quickly out-grows the possibility of effectively expressing the design in one class diagram. Using class categories and partitioning allows the system to be grouped into higher level logical pieces. As with the original class diagram, the class categories will be represented by higher level diagrams with the interfaces between the class categories defined.

2. Developing an Executable Release

Developing an executable release is the final step in the Booch method. Though beyond the scope of this thesis, a short discussion on developing executable releases concludes this chapter for the sake of completeness in the description of the Booch method. The design of the T4 game in this thesis will not include this step.

Developing the executable release can be considered the ultimate validation step in the whole iterative process. The design phase focuses on identifying the lower level details of the system. Here, new classes that are not part of the analysis model, but are needed for the implementation, are added. Designing executable releases, typically involves, but is not limited to, the following steps [White, 1994].

1. Adding control classes
2. Defining new classes to support the implementation
3. Defining operations needed to carry out the implementation
4. Defining algorithms to implement the operations
5. Providing implementations of the relationships in the analysis model
6. Adding navigation paths
7. Determining the necessary access control

These tasks need not be completed in any fixed order. They will be accomplished as a natural consequence of building the executable release.

Most of the functionality of the system is placed in the domain classes. Often, functionality required by the system involves a number of classes but does not naturally belong to any particular class. Choosing to spread these operations across several classes does not generally lead to a robust design. Often, a better solution is to introduce a control class. Control classes contain the functionality that is not naturally contained in any other

classes. These classes unite objects that collaborate to implement a specific behavior. Instances of control classes are usually only temporary in the execution of the program and exist only during the execution of a specific activity.

Many operations are simple enough that a sufficient description was made in the domain analysis. These operations do simple data access in the objects, update relationships, or update attributes. More complex operations that involve several objects need further definition. Object-scenario diagrams can expose the detail of these operations. If the operation is too complex to be illustrated in an object-scenario diagram, an algorithm description can be used to provide a more complete definition. The algorithm can be explicitly defined in the operation documentation field of the class specification. Pseudocode can be used to express the algorithm, but it is best to use the implementation code, e.g. C++.

Domain analysis modeled the association between classes. These associations are inherently bidirectional from the domain view-point, but they are often only traversed in one direction in the implementation. In design, navigation paths are added to the associations. In the C++ language, an example of a one way traversal of an association could be to use a pointer to access the data item.

In domain analysis, limiting the visibility or access to classes was not a concern. It could be assumed that all classes had access to every other class's relationships and attributes. One of the basic cornerstones of object-oriented design though, is the idea of encapsulation and data hiding. Encapsulation and data hiding enhance code reliability, maintainability, and code reuse. Along these lines, visibility and access to the internal workings of a class should be limited to the minimum amount necessary to satisfy the

requirements for the class. An added benefit of limiting access and visibility is that this fire-wall structure minimizes the effect of code changes because code changes are often localized to only one class. Limiting visibility and access can be accomplished using the specifications and class diagrams of the Booch model.

This completes the description of the Booch method of object-oriented design. A complete treatment of this method may be found in Grady Booch's textbook [Booch, 1994].

IV. TACTICAL TIC-TAC-TOE DEVELOPMENT

This chapter discusses the actual object-oriented design process for the T4 game. It follows a structure similar to the section describing the Booch method of object-oriented design.

A. REQUIREMENTS ANALYSIS

Requirements analysis for the T4 design began with detailed discussions with the user, in this case Professor Gary Porter, on required functionality of the T4 game. Detailed briefs were given by Professor Porter on the original implementation of the game using HyperCard for the Macintosh computer and on how the game was used for research in the CC4103 class projects. All available materials regarding the T4 game including previous theses, papers, briefing materials, and HyperTalk code listings were gathered as resource materials. Professor Porter also identified new requirements for the game which include the ability for networked game play and the need to use a World Wide Web browser as the common player interface to achieve platform independence. Although in many cases requirements analysis can require a long dialogue between the developer and the user, this was not the case for the T4 project. Because of the previous implementation of T4, the user had a firm grasp of all of the existing requirements with the addition of a few new requirements which were easily conveyed to the developer.

A system charter [Appendix A] was developed based on the T4 brief and previous papers which discuss the T4 functionality. The system charter is a very general listing in bullet format of what tasks the T4 game will be able to perform in order to meet the user's

requirements. Here, the need is to be able to provide the basic game functions such as team play, provision for artificial players, information delays, scoring, and record keeping. In addition to these, the needs for networked play, platform independence, and the WWW interface were listed. After review by the user, the first draft of the system charter was revised to reflect more accurately the desires of the user. This feedback loop between the user and developer is in keeping with the iterative concept that is a keystone of the Booch method and which took place throughout the T4 design process.

After the system charter was developed, a system function statement [Appendix B] was constructed to list the functionality required to implement the program's capabilities identified in the system charter. The system function statement is in a bullet format similar to the system charter, but it lists the primitive operations needed to implement the general capabilities. An example of the functions identified for the T4 game are to tally Tic-Tac-Toes, to tally mission accomplishments, to tally conflict resolutions, and to log moves. These functions support the capability to maintain a game log for replay and analysis as is identified in the system charter.

This requirements phase was revisited and the documents updated as new requirements were identified or redefined in the later stages of development. At the completion of the requirements phase for the T4 game, the development moved into the domain analysis phase.

B. DOMAIN ANALYSIS

This section describes the domain analysis process for the T4 project. The discussion will include class, relationship, operation, attribute, and inheritance definition along with validation and iteration.

1. Defining Classes

In order to begin defining classes for the T4 design, a narrative system description was created [Appendix C]. The narrative is written as an article describing what the T4 game does. The narrative provides a starting point for identifying the system classes. The nouns of the narrative were underlined and from these nouns a list of possible system classes was created. Some of the words representing candidate classes from this list were easily eliminated because they could not possibly represent a class of the system or they could be eliminated for vagueness or ambiguity. Examples of words like these are calculation, this, Tic-Tac-Toes, and moves. These candidate classes were removed from consideration. Words that did have a possibility for selection as a class were words like player, square, team, game board, score, log, and side. Player, square, game board, and log were all selected as viable representations of system classes. Score was eventually downgraded from a class to a data item, and side was converted to team because the name is more descriptive. Eventually the team class itself was eliminated because it did not contain enough data and functionality itself to warrant existence as a class and eventually became a data item of the player class.

Picking classes from the narrative description is not sufficient to identify all of the classes necessary to describe the whole system. To identify more obscure classes, the use

cases, or scenarios, in which the program will be used were analyzed in order to identify objects that participate in a scenario and the roles that they play in the system. By examining the domain from this aspect it was apparent that the player class must be divided into a human player class and a computer player class. The need for a game engine class and an interface class are so abstract that it is unlikely that they could ever have been identified simply by examining a narrative description of the proposed system.

Concise meaningful names were chosen for each class. The names were chosen in accordance with C++ programming syntax requirements. The naming convention used requires that the name begin with a lower case character. Follow-on words in the name begin with an upper case character to enhance readability. The name is suffixed with the word "Class" to distinguish the class name in the program source code from actual instances of the class (objects). An example of this naming convention is "gameEngineClass".

The classes for the T4 game are contained in the cloud compartments (class symbols) in the class diagram [Appendix D] and are also listed in the class specifications [Appendix E].

2. Defining Relationships

In order to start developing the behavior of the classes in the T4 system, their relationships must be identified. Recall that the two types of relationships are association which is a semantic dependency and aggregation which means that one class is part of the other class. The relationships discussed in this section may be viewed in the class diagram [Appendix D] and the class specifications [Appendix E].

In the class diagram, the game engine class relates to the most other classes and best

characterizes a hub. The description of class relationships will begin here and work outward along the spokes from the hub. The game engine class has a game board class which has a square class. This is a nested aggregate relationship between the three classes.

The relationships also have cardinality associated with them. The game board can have from one to many square class objects, but the square class must have one and only one game board. The same type of cardinality exists for the relationship between the game engine class and the game board class as exists for the relationship just described. The game engine can handle more than one game board object, but the game board can belong to one and only one game engine. The cardinality is specified with the relationships in the appendices. Descriptions of cardinality soon become repetitive and tedious and will not be explicitly described in the main body of the thesis for the relationships that follow.

The game engine class is associated with the log class which has a move class and a turn result class. The game engine also has a configuration class which contains all of the configuration parameters that the user uses to customize game play. The game engine is also associated with the interface class which has a controls class and a display class. The controls class accepts input from the user and the display class provides output to the user's monitor. Finally, the game engine associates with the player class. The player class is subdivided into a human player class and a computer player class. The computer player class is further subdivided into strategy classes. The human player class also has a direct association with the interface class. The subdivided player class is discussed further in a following section on inheritance.

In the Booch method the relationships are named something meaningful that

expresses the nature of the relationship. Examples of relationship names in the T4 design are "simulation control" for the relationship between the game engine class and the interface class and "basic unit' for the relationship between the square class and the game board class. Note that "basic unit" implies more of a target/source relationship where "simulation control" implies more of a bidirectional relationship.

3. Defining Operations

Definition of operations for the T4 classes took place sporadically instead of one concentrated effort to define the operations for the whole collection of classes in the system at once. It was found that required operations were realized little by little as work was being done on other parts of the design. In these instances, using a CASE tool proved very beneficial. The CASE tool allows the designer to quickly pop over to a class specification to add an operation with only a few mouse clicks. After entering the operation in the class specification, the designer can then return to the task that was underway when the operation came to mind without ever losing the original train of thought.

Operations for the T4 classes were initially discovered in much the same way as classes were. Stepping through scenarios based on the system function statement and identifying required operations along with the classes responsible for those operations provided a good first cut at operation definition. To illustrate some of the considerations for selecting the operations for the classes in the T4 project, a discussion of some of the T4 classes and their operations is included below.

The game board class is not a very complicated class and serves as a good example class. This is evident when one thinks of the operations that must be completed in order for

the game board to function. Four operations were identified for the game board class. The operations are expressed in the Booch model as function names since that is what these operations will become when they are implemented in code. Operation specifications are found in Appendix E. The first operation identified was the "initializeBoard()" operation. It was immediately clear that in order to begin a game, the game board has to be in some start state with the type of board defined. This object cannot create itself. An operation must be defined that creates the board and brings it to the initial condition. The next operation identified is that of identifying when moves from two players are in conflict with one another. This is accomplished with the "isConflict()" operation. This operation will be used each time a move is made. Assuming that there is no conflict or that the conflict has been resolved, there needs to be an operation that assigns a players mark to a particular square. This operation is called take "takeSquare()". Finally there needs to be a method of determining when the game is over, that is, no more open squares are left on the board. It makes sense that an operation of the game board class should check for this condition. This operation is called "isGameOver()" and is also used in every turn of the game.

The game engine class has numerous operations defined in the specification in Appendix E, but two operations of interest are noted here. The player class must have access to a game engine operation that allows the player to submit a move to the game engine. This operation is called "submitMove()". Although one's first inclination might be to include this operation in the player class since it is the player that submits the move, a more logical design is for the player to call an operation of the game engine class using the move as the argument to the operation. On the other hand, some of the operations in the game engine

class are intended only to be used by the game engine class. An example of this is the "resolveConflict()" operation. When the game engine is informed that moves are in conflict by checking the game board, the game engine must have a method for resolving that conflict. That method is contained in the "resolveConflict()" operation.

4. Defining Attributes

The first step in defining attributes for the T4 classes was to look at each class individually to identify any obvious attributes. Some attributes are readily apparent. By examining the player class it was clear that the game program would need a device for keeping track of the players during game play. This illustrated the need for a player identification number. The player identification number will most commonly be used in conditional statements in the program. Integers tend to work better than real numbers in conditional statements, so the player identification number was chosen to be an integer data type. The data item has to be uniquely identifiable in the source code of the program, so a name for the data item must be chosen. The name for the integer variable that represents the player identification number was chosen to be "playerNumber". In the specification for the player class an integer called "playerNumber" is defined, with a default value of zero in keeping with safe programming practices.

The narrative system description was also examined to see if class attributes could be determined from the verbiage of the description. The description discusses a double board game and the need for a game board border to separate the individual boards. A border does not warrant status as a class in its own right, but the game certainly requires information regarding the border in order to implement a double board game. The game board size is

specified in height and width by integer numbers representing squares. Integer numbers were used because it is impossible to use less than a whole square in the context of the T4 game. Since the game board is defined in integer units, it follows that the border's position on the board should be represented as an integer data type. No other information is required about the border. The game engine only needs to know where the border is. In keeping with the previously used naming convention, "vertBorder" was chosen as the name of this integer data type in the game board class. The name "vertBorder" was chosen in anticipation of T4 upgrades where more complicated boards with horizontal boarders also are used.

Finally, many attributes were identified when working on other tasks which illustrated the need for a previously undefined attribute. When working out a scenario for processing game moves the need for a game turn counter became readily apparent. Since the game moves are processed by the game engine, the game engine class was the most likely candidate as a host for the turn counter. Since there are no fractional turns, the game turn counter was defined as an integer and named "currentTurn".

All other attributes identified in the T4 development process are defined in the class specification.

5. Defining Inheritance

Inheritance is useful in consolidating traits that are common to more than one class into one superclass. The subclasses then derive these traits from the parent class. Inheritance greatly enhances code reuse and readability.

Domain analysis for the T4 game development revealed excellent candidates for applying inheritance in the human player and simulated player classes. It is immediately

evident that these two classes have traits in common. It was also evident that the differences in the implementation of these classes precluded grouping them into one class. The obvious design choice was to group these similar traits into a player superclass while leaving the human player and simulated player specific traits in their respective classes. This arrangement allows these classes to inherit the common traits from the parent player class.

The traits that the human player and simulated player classes have in common include the player number, mission objective and player name attributes. They also have the initialize player and move prompt operations in common. The inheritance relationship is expressed in the class diagram as arrows leading from the target class to the source class. The arrows show from where the inherited traits come. These relationships are shown in the T4 class diagram [Appendix D].

Notice that the player class cloud compartment contains a symbol that is the letter "A" surrounded by an inverted triangle. This symbolizes that the player class is an abstract class. An abstract class is never instantiated in a program, that is, the T4 game will never have a generic player class object. An abstract class is only used as a vehicle for storing traits that will be inherited by other classes. These traits exist in the program only in instances of the human player and simulated player classes via inheritance.

During the domain analysis, no other classes exhibited enough common characteristics to warrant the application of inheritance.

6. Validation and Iteration

Validation and iteration of the T4 design began with construction of scenario diagrams for some simple processes in the operation of the T4 game. An exhaustive list of

scenario diagrams for all of the functionality of the T4 game could not be completed due to time constraints and the diverse configurations available in T4. This subsection discusses some typical scenarios in the T4 model.

The first scenario examined was the process from starting a game. When the game program is run, the game engine object prompts the user for the requested operation by calling the "whatNow()" function of the interface object. The user causes the interface object to call the "newGame()" function of the game engine. The game engine initializes a game board and initializes the players. The game engine then begins a game log by calling the "beginLog()" function which sets up the log to receive entries. Finally, the game engine begins game play by prompting the players for their first move with a call to the "movePrompt()" function. This completes the start game scenario.

The game play scenario picks up where the start game scenario ended. The game engine prompts the players for a move. The players submit their moves by calling the "submitMove()" function using the move as an argument for the function call. The game engine checks the move to see if it is a legal move. The game board is queried to determine if a conflict exists as a result of the moves. If required the "resolveConflict()" operation is executed. The "takeSquare()" function is used to mark the appropriate squares as occupied by a player. The cycle is completed by prompting the players for another move.

More scenario diagrams are found in Appendix F. This completes the domain analysis portion of the T4 development.

C. DESIGN

The design phase consists of defining the initial architecture and developing an executable release. Defining the initial architecture is the only portion of the design phase that is within the scope of this thesis. Portions of the initial architecture for the T4 game are discussed here.

The first step of defining an architecture is to group classes into class categories. Class categories are a filling method for grouping classes that participate in a major portion of the program. Class categories assist future developers in finding portions of the design for reuse or modification. In large enterprise size projects the designs are so complicated that class categories are a must. The T4 game however, provides one basic service (game play) and so few classes that it is all grouped into one class category called "T4 Game" for compliance with the Booch method. It is conceivable that future versions of the game could contain enough functionality, for example, online analysis, that it may someday warrant separation into more class categories.

The last part of the architecture definition to be discussed here is the choices for service software. Service software is any support or environment software that the program requires for implementation. Service software can include the operating system, graphical user interfaces, database managers, and device interfaces. The intent of this design is to use the Unix operating system. C++ code can generally be written to enable portability across a variety of Unix platforms. The graphical user interface is intended to be a nonspecific WWW browser that supports the Java programming language. A database manager is not required in this version of T4 though, future versions may require such services. Required

device interfaces will be handled implicitly by platform-specific compilers.

This concludes the object-oriented design of the T4 C4I simulation game.

V. CONCLUSIONS

A. RESEARCH QUESTION RESULTS

Object-oriented design can be used to redesign a non-object-oriented game program as evidenced by this thesis.

The CASE tool (Rational Rose C++ in this case) was extremely useful in redesigning this program. The object-oriented method relies heavily on diagraming. With the practice of iterative design, all of the diagraming would quickly slow the process to a crawl without the CASE tool.

A greater level of detail is required than was reached in the thesis to begin coding the program. More detail in operation and attribute specifics will be required.

An attempt at building in flexibility for future evolution of the software was made. For example, a double game board under the current requirements only consists of two three-by-three single game boards positioned side-by-side. The game board class however has the flexibility so accept different sized boards with a border positioned anywhere the user specifies. As long as all of the game algorithms are designed with general rules which do not depend on a specific size, this flexibility will remain in the design.

The current version of HTML does not provide the functionality to permit use of a WWW browser as a networked game interface, because it does not allow the host game engine to directly prompt the player for action. It is suspected that the Java language will provide this functionality and that the WWW browser will be an effective interface for the networked game.

B. SUMMARY

The Booch method proved to be a very usable method for object-oriented design. This method and object-oriented design in general is not as intuitive as many texts imply, but with a good tutorial and a little experience with an object-oriented language such as C++ a designer can successfully create a viable object-oriented design. For the beginner, a good CASE tool is highly recommended, if for nothing else than enforcing the rules for a particular design method.

The design is ready for the final steps of the Booch method which include determining access control, adding control classes, and defining algorithms to implement all of the operations. A firm foundation has been established for continuation of the T4 migration project.

C. RECOMMENDATIONS

It is recommended that the design be continued using the Rational Rose C++ CASE tool. This CASE tool provides functionality for configuration management, team development, code generation, and reverse engineering.

Future versions of the T4 game should include some basic analysis capability to allow remote users to benefit from the educational opportunities provided by the game. Future versions should also consider portability to PC platforms as they take over more of the computer market share currently dominated by high power workstations.

Finally, the T4 game should be made available on the Navy Online web site to maximize participation by, and education of, all DOD members.

APPENDIX A: SYSTEM CHARTER

System Charter Statement

The Tactical Tic-Tac-Toe game will be able to:

- Provide an open system, networked T4 game environment for one or more players
- Allow players to play other players, real or simulated
- Allow the computer to play itself (closed form simulation)
- Allow single or double board games
- Allow two player or team games
- Allow for information delays (tactical, communication, or area)
- Allow for assignment of up to four mission objectives from a set of eight
- Allow umpire or player full configuration of game parameters prior to commencement of play
- Keep play logs for replay and analysis.
- Keep track of player's demographics and scores
- Allow game play in LAN and WAN environments or over the internet on a variety of machines using WWW/Java interface

APPENDIX B: SYSTEM FUNCTION STATEMENT

System Function Statement

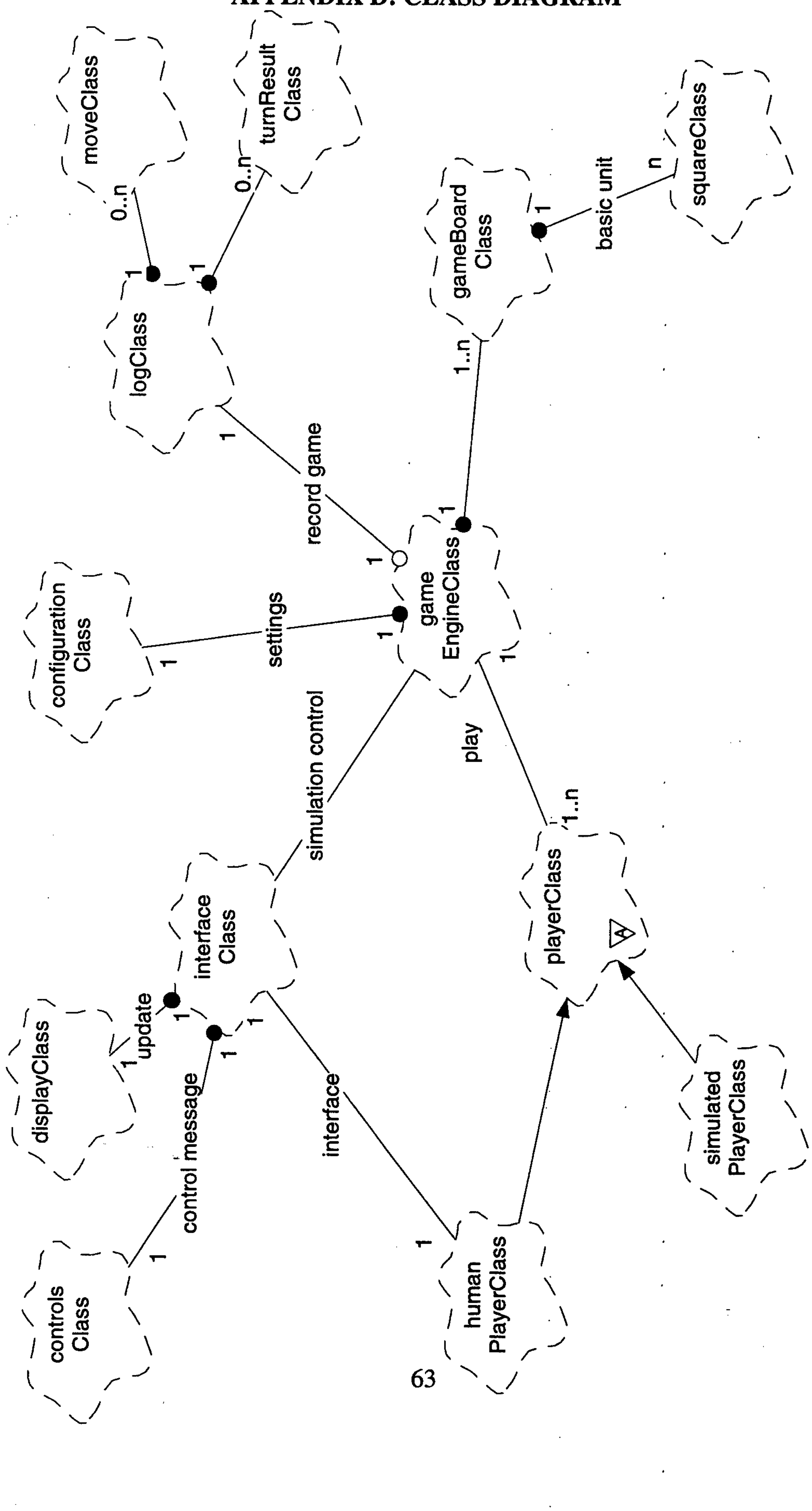
- Register new players
- Query players for game preferences
- Provide artificial players
- Customize artificial players
- Assign missions
- Provide game engine
- Display game boards
- Process moves
- Mediate conflicts
- Restrict information
- Restrict communications
- Provide communications (between team members)
- Update game boards
- Tally Tic-Tac-Toes
- Tally mission accomplishments
- Tally conflicts
- Log moves
- Export data
- Score games
- Rematch opportunity
- Display output
- Record moves, game results, attributes, and configurations to file

APPENDIX C: NARRATIVE SYSTEM DESCRIPTION

Narrative System Description

Tactical-Tic-Tac-Toe (T4) is based on the traditional game of Tic-Tac-Toe. In order to win a player tries to get more Tic-Tac-Toes than the opponent instead of trying to get the first Tic-Tac-Toe. Instead of taking turns in choosing moves, both player's moves are submitted simultaneously each turn. If opponents select the same square, a conflict occurs which is resolved by a random probability calculation. The winner of this calculation takes the square with the loser taking no square. The game can be played against another player, in teams of two against another team, or against the computer. Variations of the game allow for double games to be played where simultaneous moves are made on side-by-side boards. In the double game, crossover Tic-Tac-Toes can occur across the boundary between the two boards. Another variation of the game is to delay a player's knowledge of the opponent's moves. When this occurs a player only finds out the opponent's moves after a specified number of turns has elapsed. This delay allows for the possibility of a delayed conflict where a player unknowingly selects a square that the opponent already occupies. In this case the player who occupies the square first always wins the conflict. In the case of a normal conflict, the delay is overridden and the player knows immediately of the opponents move. Another variation is to allow team play where a pair of players represents a particular side with varying degrees of cooperation allowed. Yet another variation is assignment of victory or survival mission goals which can be by game board areas. The game will have the capability to play multiple players over a network. The game will also have an artificial intelligence module that can play against real players or itself in a fully automated mode. The closed form simulation mode's simulated players can be configured to have with different player traits. The game outputs results including scores and configuration parameters, allow for replay and postgame analysis, contains all game play information, and offer a chance for rematches.

APPENDIX D: CLASS DIAGRAM



Category name:
T4 Game

APPENDIX E: CLASS SPECIFICATION

Global: No
Subsystem: none

Class name:
gameBoardClass

Category: T4 Game
Documentation:
Definition:
A game board is two deminsional array of square objects on which the T4 game is played. The board may have a boundary which divides player areas of responsibility during multiplayer games

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:

squareClass basic unit
Definition:
The basic part of the game board is the square. A game board has several squares but a square belongs to only one game board.

Operations:

InitializeBoard
isConflict
takeSquare()
isGameOver

Private Interface:
Attributes:

height : int = 3
Definition:
This is the vertical parameter for a 2D array that represents the game board.

width : int = 3
Definition:
This is the horizontal parameter for a 2D array that represents the game board.

vertBorder : int = 0
Definition:
This serves as the dividing line for multible board games.

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

InitializeBoard

Public member of: gameBoardClass
Documentation:
Definition:
Constructs a game board for a new game according to game parameters.

Concurrency: Sequential

Operation name:

isConflict

Public member of: gameBoardClass
Documentation:
Definition:
Check players moves for conflict.

Concurrency: Sequential

Operation name:

takeSquare()

Public member of: gameBoardClass
Return Class: moveResult
Arguments:
int playerID =0
Documentation:
Also as an argument this function needs the probability for a player in conflict resolution.
Definition:
Member function takeSquare() returns the result of the move (success or fail, ID of occupant

Concurrency: Sequential

Operation name:

isGameOver

Public member of: gameBoardClass
Documentation:
Definition:
Checks for game over condition.

Concurrency: Sequential

Class name:

squareClass

Category: T4 Game
Documentation:
Definition:
A square is one element of a game board. A square can be empty or occupied by a player. A square's locality in relationship to any border is relevant. A square's status may or may not be known to all players

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Private Interface:
Attributes:

occupant : int = 0
Definition:
Occupant is the ID number (integer) of the player who occupies the square or zero if the square is empty.

turnOccupied : int = 0
Definition:
Records the turn in which a square was occupied. Used for delayed intelligence reporting and delayed conflict resolution.

visibleToPlayer : int[n] = 0
Definition:
This is an array of integers that has as many elements as there are players in the game. It specifies whether a player can see if the square is occupied or not.

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
logClass

Category: T4 Game
Documentation:
Definition:
A log is a record of the moves and occurances that take place during turns in a game. The log allows for game replay and analysis. The log will include player moves, the outcome of conflicts, and the new information presented to each player during each turn.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:
turnResultClass

moveClass

Operations:

beginLog
recordMove
endLog
openLog
closeLog
readNextMove

Private Interface:
Attributes:

logID : char[8] = 0

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

beginLog

Public member of: logClass
Return Class: int
Documentation:
Definition:
The beginLog function is used at the beginning of a game to initialize a new game log. This function creates the log and assigns a log ID.

Concurrency: Sequential

Operation name:

recordMove

Public member of: logClass
Return Class: int
Documentation:
Definition:
Records a move and the results of the move for a game in progress.

Concurrency: Sequential

Operation name:

endLog

Public member of: logClass
Documentation:
Definition:
The endlog function closes the log and writes the log to a file.

Concurrency: Sequential

Operation name:
openLog

Public member of: logClass
Return Class: int
Documentation:
Definition:
The openLog function opens a log file (read only) for game replay.

Concurrency: Sequential

Operation name:
closeLog

Public member of: logClass
Return Class: int
Documentation:
Definition:
The closeLog function closes an existing game log. Used in conjunction with openLog.

Concurrency: Sequential

Operation name:
readNextMove

Public member of: logClass
Documentation:
Definition:
This is an iterator used to move through an existing log file when replaying a game.

Concurrency: Sequential

Class name:
gameEngineClass

Category: T4 Game
Documentation:
Definition:
The game engine is the logical engine that actually mediates the play of the game. It prompts players for moves, resolves conflicts and illegal moves, reports results depending on player constraints, and determines victory or defeat for players depending on mission objectives. The game engine keeps a record in the log and provides for formatted data output.
Questions:
Does the game engine need to provide a method of planning and communication between team members in a networked game?

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Associations:

<no rolename> : playerClass in association play
<no rolename> : interfaceClass in association simulation control

Public Uses:

logClass record game

Public Interface:

Has-A Relationships:

configurationClass settings
Definition:
The game settings are defined in configuration.

gameBoardClass

Operations:

newGame
isValidMove
resolveConflict
endGame
gameScore
registerPlayers
queryParameters
submitMove

Private Interface:

Attributes:

currentTurn : int = 0
Definition:
This is the master turn counter against which, all information delays are measured.

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

newGame

Public member of: gameEngineClass
Documentation:
Definition:
Queries user for game parameters. Sets up a new game board. Initializes players.

Concurrency: Sequential

Operation name:
isValidMove

Public member of: gameEngineClass
Documentation:
Definition:
Checks to see if player is attempting to move into a legal square.

Concurrency: Sequential

Operation name:
resolveConflict

Public member of: gameEngineClass
Documentation:
Definition:
Used to determine the results of a conflict when players attept to occupy the same square.

Concurrency: Sequential

Operation name:
endGame

Public member of: gameEngineClass
Documentation:
Definition:
Stops the game when the game is complete. Closes the game log. Sends results to players.

Concurrency: Sequential

Operation name:
gameScore

Public member of: gameEngineClass
Documentation:
Definition:
Calculates the score for each player based on outcome and configuration. Probably called by end game.

Concurrency: Sequential

Operation name:
registerPlayers

Public member of: gameEngineClass
Documentation:
Definition:
Assigns player ID number and takes player information.

Concurrency: Sequential

Operation name:
queryParameters

Public member of: gameEngineClass
Documentation:
Definition:
Queries user for game parameters at game start.

Concurrency: Sequential

Operation name:
submitMove

Public member of: gameEngineClass
Documentation:
Definition:
Submits the player's move to the game engine.

Concurrency: Sequential

Class name:
configurationClass

Category: T4 Game
Documentation:
Definition:
The configuration provides parameters for game play to the game engine. Configuration items include type of game, game board size, game board divisions (boundaries), number of players, player's mission objectives, areas of responsibility, and player's constraints.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

humanPlayerClass

Category: T4 Game
Documentation:
Definition:
This class represents a real player in the game.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: playerClass
Associations:

<no rolename> : interfaceClass in association interface

Public Interface:
Operations:
gameResult

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

gameResult

Public member of: humanPlayerClass
Documentation:
Definition:
Prompts player that game is over.

Concurrency: Sequential

Class name:

displayClass

Category: T4 Game
Documentation:
Definition:
The display provides for game board presentation to the user and allows the game to prompt the user for input. The display should be able to receive standardized output from the API.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

controlsClass

Category: T4 Game
Documentation:
Definition:
Things the user uses to control the game.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

simulatedPlayerClass

Category: T4 Game
Documentation:
Definition:
The computer player has an A-I module that provides the algorythm for an computer player to decide on moves.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: playerClass
Private Interface:
Attributes:

regGamePlan
Definition:
A two dimensional array containing the regular game plan.

crossoverGamePlan
Definition:
A two dimensional array containing the crossover game plan.

cellGamePlan
Definition:
A two dimensional array containing the cell game plan.

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

playerClass

Category: T4 Game
Documentation:
Definition:
A player is a participant in a game. A player can be a real person or a computer player. A team has mission objectives. A player may or may not have a team mate. Team membership is indicated as a part if the player ID number. A player may operate under a series of constraints. A player has a customized view of the game board that may not be the same as another player's view.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Associations:

<no rolename> : gameEngineClass in association play

Public Interface:
Operations:
initializePlayer
movePrompt

Private Interface:
Attributes:
playerNumber : int = 0
Definition:
This number identifies a player for the duration of the game. Moves, displays, and log entries are identified by this number.

missionObjective
Definition:
Records a players mission objectives according to the initial game setup. Used in postgame scoring.

playerName : char[20]
Definition:
Idenities players by name. Artificial players will use the artificial strategy type as the name.

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:
initializePlayer

Public member of: playerClass
Documentation:
Definition:
Creates new, human or artificial, players for a new game.

Concurrency: Sequential

Operation name:

movePrompt

Public member of: playerClass
Documentation:
Definition:
The game engine uses this to prompt the players for the next move.

Concurrency: Sequential

Class name:

interfaceClass

Category: T4 Game
Documentation:
Definition:
The interface class acts as an intermediary between the game program and the user. It accepts input from the controls and sends output to the display. The interface class is highly dependent on the tpye of environment the program will run in.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Associations:

<no rolename> : humanPlayerClass in association interface
<no rolename> : gameEngineClass in association simulation control

Public Interface:
Has-A Relationships:

controlsClass control message
displayClass update

Operations:

updateDisplay
signalEndGame
whatNow

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

updateDisplay

Public member of: interfaceClass
Documentation:
Definition:
Prompts interface module to update players display.

Concurrency: Sequential

Operation name:
signalEndGame

Public member of: interfaceClass
Documentation:
Definition:
Tells player that game is over and posts results

Concurrency: Sequential

Operation name:
whatNow

Public member of: interfaceClass
Documentation:
Definition:
Prompts user for action desired (game play or game replay or stop program) at program startup or between games.

Concurrency: Sequential

Class name:
moveClass

Category: T4 Game
Documentation:
Definition:
The move class is a structure that contains the move information for each turn.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
addMove()

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

addMove()

Public member of: moveClass
Return Class: boolean
Documentation:
Definition:
Adds a move record.

Concurrency: Sequential

Class name:

turnResultClass

Category: T4 Game
Documentation:
Definition:
A structure that contains the turn results from each moves. These records show the result of conflict resolution>

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
addResult()

State machine: No
Concurrency: Sequential
Persistence: Transient

Operation name:

addResult()

Public member of: turnResultClass
Documentation:
Definition:
Adds a turn result record.

Concurrency: Sequential

Association:

play

Documentation:
Definition:
Play describes the two way interaction between the player class and the game engine class.

Derived: No
Direction: <non-directional>
Association Class: none

Role:
Class: gameEngineClass
Cardinality / Multiplicity: 1
Navigable: No
Aggregate: No
Static: No
Friend: No
Access: Public
Containment: Unspecified

Role:
Class: playerClass
Cardinality / Multiplicity: 1..n
Navigable: No
Aggregate: No
Static: No
Friend: No
Access: Public
Containment: Unspecified

Association:

interface

Documentation:
Definition:
Interface describes the direct interaction between the user and the player class. This interaction is necessary for the submission of moves.

Derived: No
Direction: <non-directional>
Association Class: none

Role:
Class: humanPlayerClass
Cardinality / Multiplicity: 1
Navigable: No
Aggregate: No
Static: No
Friend: No
Access: Public
Containment: Unspecified

Role:
Class: interfaceClass
Cardinality / Multiplicity: 1
Navigable: No
Aggregate: No
Static: No
Friend: No

Access: Public
Containment: Unspecified

Association:

simulation control

Documentation:

Definition:
This association allows a user to set up simulation runs with computer players where the user does not have to be a player.

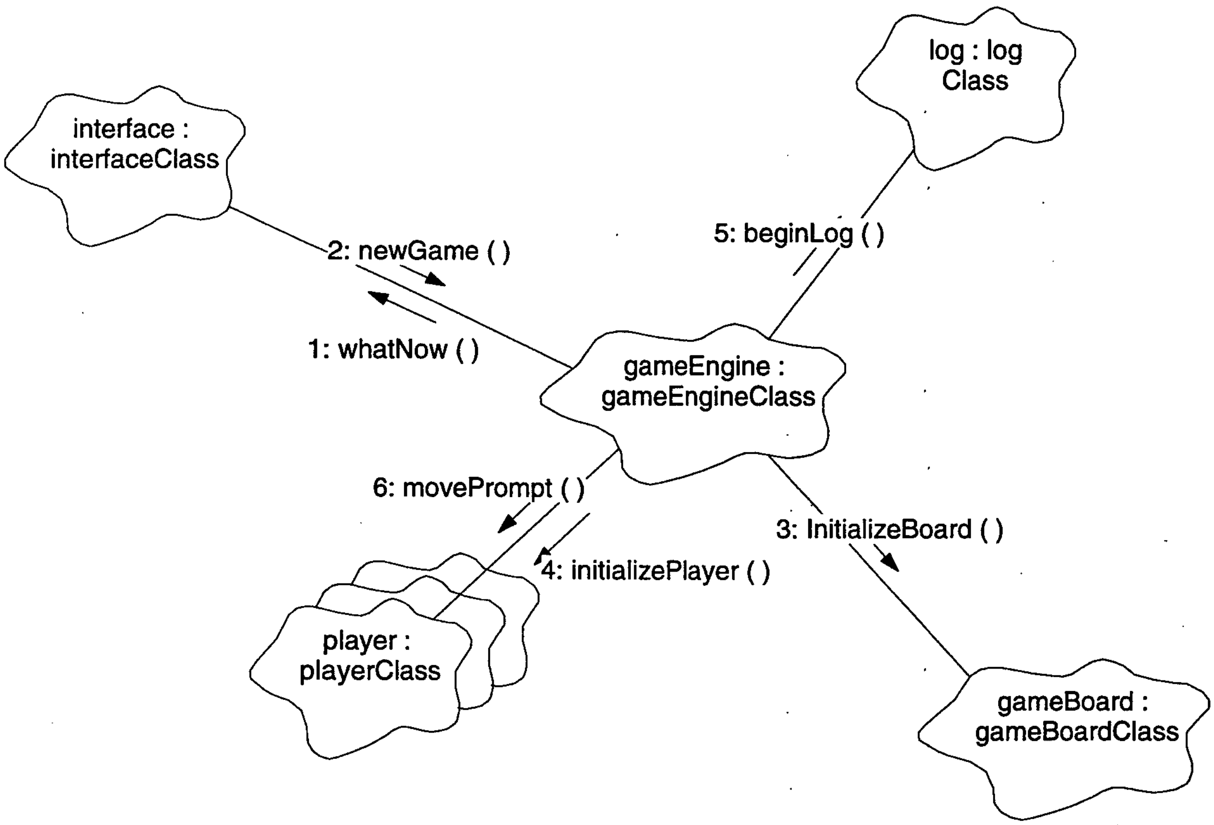
Derived: No
Direction: <non-directional>
Association Class: none

Role:
Class: gameEngineClass
Cardinality / Multiplicity:
Navigable: No
Aggregate: No
Static: No
Friend: No
Access: Public
Containment: Unspecified

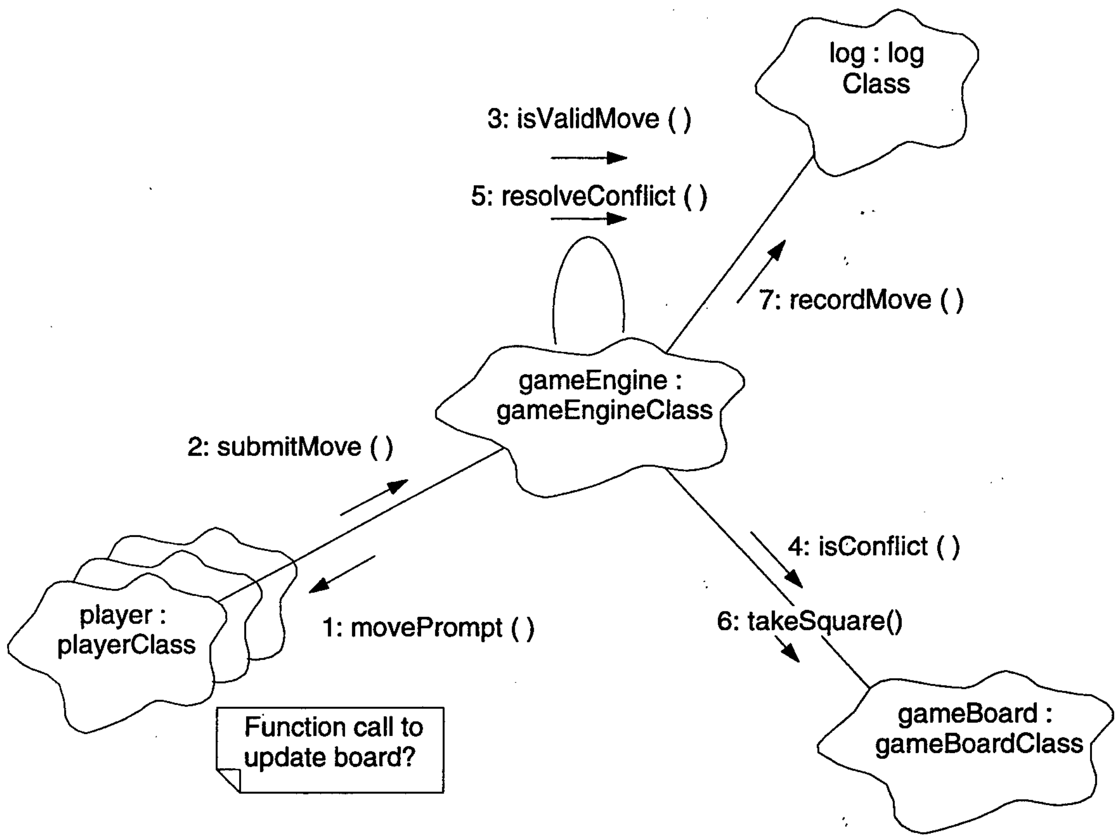
Role:
Class: interfaceClass
Cardinality / Multiplicity:
Navigable: No
Aggregate: No
Static: No
Friend: No
Access: Public
Containment: Unspecified

APPENDIX F: SCENARIO DIAGRAMS

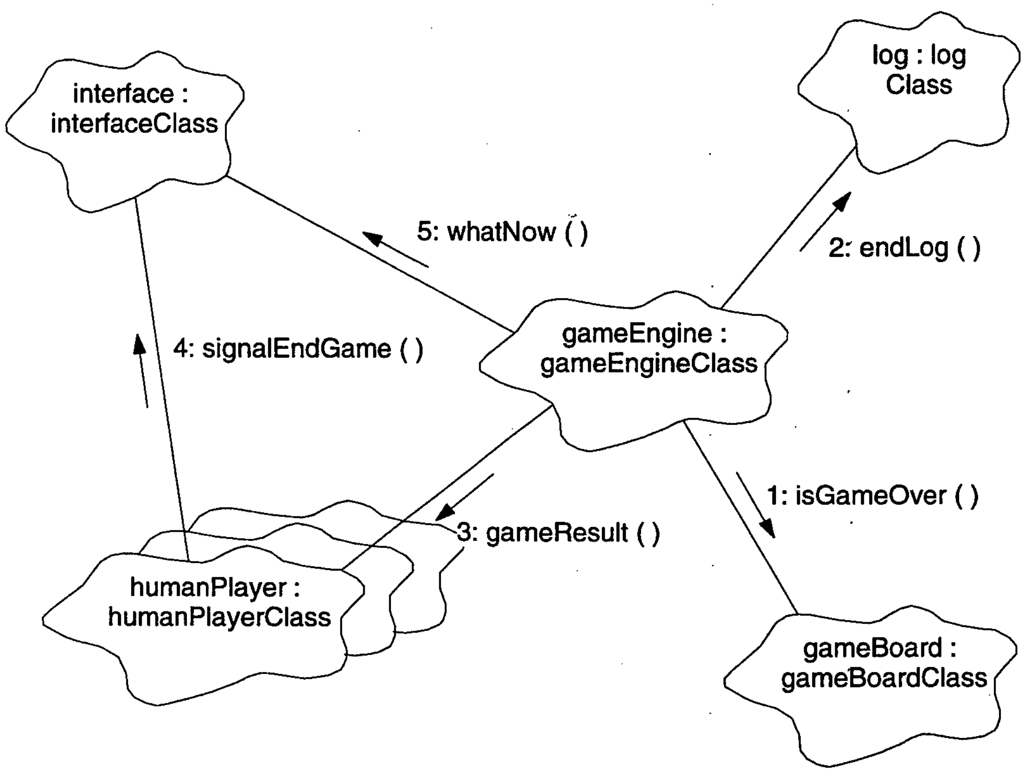
Start Game Scenario



Game Play Scenario



End Game Scenario



LIST OF REFERENCES

Booch, Grady, "Object-Oriented Analysis and Design with Applications", The Benjamin/Cummings Publishing Company, Inc., 1994.

Deitel, H. M., Deitel, P. J., "C++ How To Program", Prentice-Hall, Inc., 1994.

Pohl, Ira, "Object-Oriented Programming Using C++", The Benjamin/Cummings Publishing Company, Inc., 1993.

Sommerville, Ian, "Software Engineering", Addison-Wesley Publishing Company Inc., 1992.

White, Iseult, "Using the Booch Method", The Benjamin/Cummings Publishing Company, Inc., 1994.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, CA 93943-5101	2
3.	Director for Command, Control, and Communications Systems Joint Staff Washington, DC 20318-6000	1
4.	Prof. Gary Porter, Code CC/PO Naval Postgraduate School Monterey, California 93943-5000	10
5.	Prof. Michael Sovereign, Code CC/SM Naval Postgraduate School Monterey, California 93943-5000	1
6.	Chairman, Code CC Naval Postgraduate School Monterey, California 93943-5000	1
7.	Lt. Todd L. Lennon 404 Division St. Greenville, TX 75401	2